INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

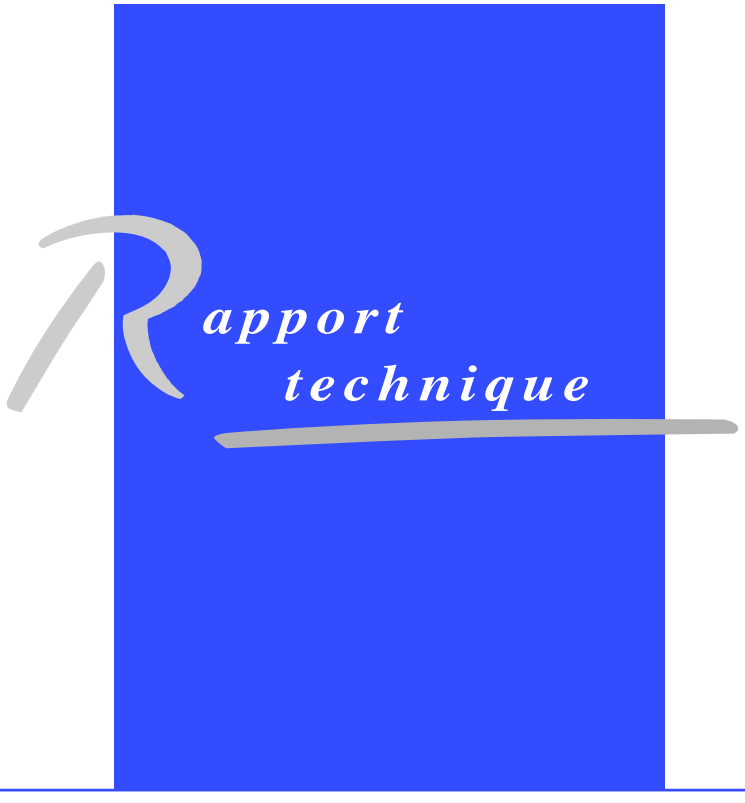
Shrimp *User Guide. A Fast Mesh Renumbering and Domain Partionning Method*

Adrien Loseille and Frédéric Alauzet

N° 0362

February 19, 2009

Thème NUM



ISSN 0249-0803 ISRN INRIA/RT--0362--FR+ENG

Shrimp User Guide. A Fast Mesh Renumbering and Domain Partionning Method

Adrien Loseille* and Frédéric Alauzet†

Thème NUM — Systèmes numériques
Projet Gamma

Rapport technique n° 0362 — February 19, 2009 — 25 pages

Abstract: This technical note describes the main features of Shrimp‡, a software that renumbers mesh entities and splits mesh domain and handle the parallelization of adaptive mesh generators. The aim of the software, the input and the output files and the list of options are given in this document.

Shrimp has been developed within the GAMMA research project at INRIA Paris-Rocquencourt.

This document describes the features of the current version: release V1.0 (January 2009).

Key-words: Domain splitting, Mesh renumbering, Parallel computing, Parallel mesh adaptation.

* Email : Adrien.Loseille@inria.fr
† Email : Frederic.Alauzet@inria.fr
‡ This software was registered with the APP under n° IDDN.FR.001.070013.000.S.P. 2009.000.10000 on february 10, 2009

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Shrimp Guide de l'utilisateur. Une méthode de renumérotation de maillage et de partitionnement de domaine rapide

Résumé : Ce rapport technique décrit les principales fonctions de Shrimp[§], un logiciel qui renumérote et partitionne/décompose des maillages et gère la parallélisation des mailleurs adaptatifs. Les fonctionnalités du logiciel, les formats des fichiers d'entrée/sortie et les options sont donnés dans ce document.

Shrimp a été développé au sein de l'équipe projet GAMMA à l'INRIA Paris-Rocquencourt.

Ce document décrit les fonctionnalités de la version courante : version V1.0 (Janvier 2009).

Mots-clés : Partitionneur de domaine, renumeroteur de maillage, Calcul parallèle, Adaptation de maillage parallèle.

[§] Ce logiciel a été enregistré à l'APP sous le numéro n° IDDN.FR.001.070013.000.S.P. 2009.000.10000 le 10 février 2009

Contents

1 Shrimp overview

1.1 Context

The efficient use of computer hardware is crucial in scientific computing to achieve high-perfomance computing. However clever an algorithm may be, it has to run efficiently on the available computer hardware. Each type of computer, from a PC to the fastest massively parallel machine, has its own shortcomings that must be accounted for when developing both the algorithms and the simulation code. Fortunately, some of the main issues can be addressed by external code, as `Shrimp`, in order to achieve good performance on the most common types of computers.

1.2 Main features

The main issues addressed by `Shrimp` are the reduction of cache misses by mesh renumbering, the improvement of shared memory parallel machine by mesh renumbering and domain splitting for parallel runs on distributed memory parallel machine. All the algorithms are based on space filling curves. It also handles parallel mesh adaptation associated with a compatible mesh generator. Currently, it is coupled with `Mmg3d`[1] and `gamanic`[2]. `Shrimp` deals with 2D triangular, 3D tetrahedra and 3D triangular surface meshes.

Mesh reordering for serial code. The Hibert-based algorithm used for mesh partitioning is very fast and gives impressive results in the speed up of serial codes. With this option, `Shrimp` aims at reducing cache misses and thus increases the serial speed of a code. A speed up of 3 is generally observed for our in-house finite volume solver `Wolf` [2] in serial on strongly anisotropic meshes. A speed-up up to 10 is observed on the adaptive mesh generator `Mmg3d` [3]. Of course, all these speed-ups include the mesh rerenumbering time.

Mesh partitionning for parallel mesh adaptation. Two options are available:

1. *Hilbert-based partitionning.* Mesh partitioning with `Shrimp` is very fast as compare to graph based algorithms. However, these two approaches aim at very different goals. The first one is specifically designed to perform mesh adaptation in parallel. In this case, the load balancing is directly performed by `Shrimp` and does not depend on the size of the interface. On the contrary, graph-based methods usually try to minimize the surface of the interface in order to minimize inter-cpu communications. For parallel adaptation, `Shrimp` can be used both on distributed and shared memory architectures.

2. *User-based partitionning.* `Shrimp` can handle user defined paritition by reading specific reference associated with each tetrahedron. For instance, this mode can separate boundary layers from a the volume mesh. The boundary layer mesh can be kept while

[1] Contact frey@ann.jussieu.fr, free for non commercial purpose
[2] Contact Paul-Louis.George@inria.fr

the volume mesh is adapted. This mode can be used to ease the visualization of huge mesh by splitting the domain or by creating a partition around areas of interest. If needed, `Shrimp` corrects the user-defined partitions in order to ensure that all parts are connex.

The following documentation is an introduction to `Shrimp`. Several examples are explained.

1.3 Languages and platform

The program is entirely written in C (C89 ANSI norm). The current version consists of about 13 000 lines of optimized source code. The code is free of any external libraries. It uses pthread for parallelization. Hence, the code is very portable and has been successfully compiled and tested on all major computer architectures (i.e., HP, IBM, Intel- based PC, etc.) and operating systems (Unix/Linux, WindowsNT, Mac OS).

Notice that `Shrimp` is not supplied with a mesh generator.

1.4 Software integration

If `Shrimp` is integrated in a software package, only the input and output routines need to be modified, for efficiency and compatibility purposes. In this context, no more than a few routines need to be modified and adapted to support the user file formats.

1.5 Distribution

An evaluation copy of `Shrimp` software for a limited period of time can be obtained by contacting the authors at INRIA :

Frédéric ALAUZET
INRIA, Domaine de Voluceau
BP 105, 78153 Le Chesnay cedex, France
Email: frederic.alauzet@inria.fr

Adrien LOSEILLE
INRIA, Domaine de Voluceau
BP 105, 78153 Le Chesnay cedex, France
Email: adrien.loseille@inria.fr

2 Input and output data

`Shrimp` requires the specification of meshes and possibly solution fields. It ouputs split or renumbered meshes and solution fields. The specification of the discrete support, *i.e.* the mesh, is done by the `mesh` format. 2D meshes, 3D surface meshes and 3D meshes can be specified. As regards solution fields, they are specified with the `sol` format.

2.1 Mesh specification

Meshes are described using the `mesh` file format. The `mesh` format describes precisely meshes and also the surface features. This format is composed of a single (ASCII or binary) file, `xxx.mesh` or `xxx.meshb`. This file contains all the information needed to describe entirely the mesh.

Its structure is organized as a series of fields identified by keywords. The blanks, "newline" or <CR> and tabs are considered as item separators. A comment line starts with the character # and ends at the end of the line. The comments are placed exclusively between the fields. The mesh file must start with the descriptor :

```
MeshVersionFormatted 2
Dimension 3 # or 2 in 2D
```

The other required fields for `Shrimp` correspond to the geometry (*i.e.,* the coordinates) and to the topology description (*i.e.,* the mesh entities). In the following tables, the term v_i indicates a vertex index, *i.e.,* the i^{th} vertex in the vertices list. The vertices are defined by their coordinates either in simple or in double precision. The reference is an integer attached to the vertex. For instance, it can represent a tag for boundary conditions or the tag of a partition. The elements inside the domain or on the boundary are defined by their list of vertices where each vertex id is given thanks to an integer. The reference is an integer attached to the element.

3D meshes:

Keyword	Card.	Syntax	Range
`Vertices`	nv	$x_i\ y_i\ z_i\ vref_i$	$\{i = 1, nv\}$
`Tetrahedra`	nt	$v_i^1\ v_i^2\ v_i^3\ v_i^4\ tref_i$	$\{i = 1, nt\}$
`Triangles`	nf	$v_i^1\ v_i^2\ v_i^3\ fref_i$	$\{i = 1, nf\}$

2D meshes:

Keyword	Card.	Syntax	Range
`Vertices`	nv	$x_i\ y_i\ vref_i$	$\{i = 1, nv\}$
`Triangles`	nt	$v_i^1\ v_i^2\ v_i^3\ tref_i$	$\{i = 1, nt\}$
`Edges`	ne	$v_i^1\ v_i^2\ eref_i$	$\{i = 1, ne\}$

Finally, the data structure must end with the keyword:

```
End
```

Let us give an example:

```
MeshVersionFormatted 2
Dimension 2

# Set of mesh vertices (x,y,ref)
Vertices
581
0.1 1.  0
0.333 12.125 0
.........

# Set of mesh triangles (v1,v2,v3,ref)
Triangles
1162
1 28 521 0
23 45 77 0
.........

# Set of mesh edges (v1,v2,ref)
Edges
212
1 28 3
28 34 3
.........

End
```

2.2 Solution or metric specification

Solution fields are described using the `sol` file format. The `sol` format describes several types of solutions (scalar, vector, tensors,...) which can be linked to different mesh entities. This format is composed of a single (ASCII or binary) file, `xxx.sol` or `xxx.solb`.

Its structure is organized as a series of fields identified by keywords. The blanks, "newline" or <CR> and tabs are considered as item separators. A comment line starts with the character # and ends at the end of the line. The comments are placed exclusively between the fields. The sol file must start with the descriptor :

```
MeshVersionFormatted 2
Dimension 3 # or 2 in 2D
```

The solutions fields for `Shrimp` are associated with the vertices of the given mesh and are

defined by the keyword `SolAtVertices`. This keyword is followed by the number of entities (here vertices) supporting the solution, the number of types and the list of solution types. The type of solutions handled by `Shrimp` can be scalar, vector or tensor fields. They are defined as follow depending on the dimension

3D solution type:

Field	Type	Syntax	Range
`Scalar`	1	f_i	{i=1,nv}
`Vector`	2	$f_i^1\ f_i^2\ f_i^3$	{i=1,nv}
`Tensor`	3	$f_i^1\ f_i^2\ f_i^3\ f_i^4\ f_i^5\ f_i^6$	{i=1,nv}

2D solution type:

Field	Type	Syntax	Range
`Scalar`	1	f_i	{i=1,nv}
`Vector`	2	$f_i^1\ f_i^2$	{i=1,nv}
`Tensor`	3	$f_i^1\ f_i^2\ f_i^3$	{i=1,nv}

where the convention for tensors is

$$\mathcal{M}_{2D} = \begin{pmatrix} f_i^1 & f_i^2 \\ f_i^2 & f_i^3 \end{pmatrix} \qquad \text{and} \qquad \mathcal{M}_{3D} = \begin{pmatrix} f_i^1 & f_i^2 & f_i^4 \\ f_i^2 & f_i^3 & f_i^5 \\ f_i^4 & f_i^5 & f_i^6 \end{pmatrix}$$

Finally, the data structure must end with the keyword:

```
End
```

Let us give an example for each solution type:

Scalar

```
MeshVersionFormatted 2

Dimension 2

SolAtVertices
581
1 1
1.
0.125
0.125
.....
End
```

Vector

```
MeshVersionFormatted 2

Dimension 2

SolAtVertices
581
1 2
0. 0.
0.3945 2.55264e-05
0.245741 1.14493e-05
.....
End
```

Tensor

```
MeshVersionFormatted 2

Dimension 2

SolAtVertices
581
1 3
70.8852 0 70.8852
72.6135 0 72.6135
63.7954 0 63.7954
.....
End
```

and a final example for several solution types associated to mesh vertices. There are three solution types. The first type is a scalar, the second a vector and the third one a scalar. In the file, the first column corresponds to the first scalar solution field. The second and the third columns correspond to the vector solution field. And the last column is associated with to the second scalar solution field.

```
MeshVersionFormatted
2

Dimension
2

SolAtVertices
25282
3 1 2 1
1 0 0 2.5
0.125 0 0 0.25
0.125 0 0 0.25
1 0 0 2.5
0.425971 0.3945 2.55264e-05 0.941469
............
End
```

3 `Shrimp` Howto

The following questions/answers give a complete overview of `shrimp`.

1. How I reorder a mesh ?

   ```
   shrimp -O 1 -in name -out name.reor
   ```

   `Shrimp` reads `name.mesh[b]`, and if it exists, `shrimp` reads `name.sol[b]` and creates `name.reor.mesh[b]` and `name.reor.sol[b]` according to the input and the corresponding solution file. Binary files are always read first. If `name.mesh` and `name.meshb` exist, then `name.meshb` is read.

2. How do I split a mesh into 8 parts ?

   ```
   shrimp -O 2 -in name -ncut 8 -out part
   ```

   From `name.mesh[b]` and solution file `name.sol[b]` (if it exists), `shrimp` creates `part.1.mesh[b]`, ..., `part.8.mesh[b]`, plus solutions files `part.1.sol[b]`, ..., `part.8.sol[b]`. Two options are available for this module:

`-nordm`	cancels the randomization for the partitioning.
`-nocor`	avoids the correction of the final partitions.

   Be careful, with the latter option, partitions may be non-connex. Without corrections and without random, the call becomes:

   ```
   shrimp -O 2 -in name -ncut 8 -out part -nordm -nocor
   ```

3. I have a mesh with different references on tetrahedra, how I create a partition corresponding to each tag ?

   ```
   shrimp -O 2 -in name -ncut 8 -out part -tref
   ```

   The partitions indices are ordered in an increasing order with respect to the references. In this case, the option `-nocor` may also be used to avoid corrections.

4. How do I gather partitions ?

   ```
   shrimp -O 3 -in part -ncut 8 -out final
   ```

   Reads `part.1.mesh[b]`, ..., `part.8.mesh[b]`, plus if solutions files exist `part.1.sol[b]`, ..., `part.8.sol[b]`. Then, it writes `final.mesh[b]` and if solution files have been read `final.sol[b]`.

5. To adapt the mesh with `Mmg3d`, I use a single command line. Can I replace it with a single command line of `Shrimp` to run it in parallel on a shared memory parallel machine?
   The answer is Yes. If the `Mmg3d` command line looks like:

   ```
   mmg3d -O 1 -in name final -m 512 -bucket 512
   ```

   Then, the equivalent `Srimp` command line to run `Mmg3d` in parallel on a shared memory parallel machine is

   ```
   shrimp -O 5 -in name -out final -mem 512 -bucket 512 -ncut 8
   -nproc 8 -sref
   ```

   Notice that the options `-bucket` and `-mem` are directly transfered to the mesh generator. The option `-sref` is used to re-encode triangle (edge in 2D) references on short integer. This option is mandatory for codes compatibility when the entity references are stored with a short int in the mesh generator. In that case, a ascii hash table of reference is outputted. It is re-read while gathering partitions.

   When the working directory is shared or exported, it may have different names according to the server (*e.g.* /net/...). To deal with this case, the option `-dir` modifies the remote directory name:

   ```
   shrimp -O 5 -in name -out final -mem 512 -bucket 512
   -ncut 8 -dir /net/form/Users/lolo/working
   ```

   whereas the command `pwd` could locally answer `working` only.

6. To adapt the mesh with `Mmg3d`, I use a single command line. Can I replace it with a single command line of `Shrimp` to run it in parallel on distributed architectures ?
   The answer is Yes. The equivalent `Shrimp` command line is

   ```
   shrimp -O 5 -in name -out final -mem 512 -bucket 512
   -ncut 8 -host -sref
   ```

   You must create a file named `host.dat` that contains the machines names along with their number of CPUs. An example of file `host.dat` is:

   ```
   2
   form.inria.fr 2
   morue.inria.fr 4
   ```

   The first line is the number of machines followed by the host names and their number of CPUs. You can use the option `-w` to write an example of `host.dat` file:

```
shrimp -O 5 -in name -out final -mem 512 -bucket 512
-ncut 8 -host -sref -w
```

The communications are done using SSH2. Consequently, the user may connect to all the listed machines in `host.dat`. To do so, one may use the following:

```
Connect without password with ssh2
  On the client host, saying picarel, type:
lolo@picarel>ssh-keygen -t rsa
lolo@picarel>cd .ssh
lolo@picarel>chmod go-r id_rsa.pub
lolo@picarel>cp id_rsa.pub authorized_keys
You need to  do  that once. (whatever the number of targeted servers).
  On  the server machine, saying form, type:
lolo@form>cd .ssh
lolo@form>chmod go-r authorized_keys2
lolo@form>echo "fingerprint" >> authorized_keys2
where fingerprint is just a cat of id_rsa.pub (of client host)
Now, you can log without password try lolo@picarel>ssh lolo@form
```

7. How I can optimize the final outputted adapted mesh ?

   When, the mesh is split in several parts for adaptation in parallel, the mesh generator preserve the interfaces. In consequence, the regions of the mesh at the interfaces of each partition are not adapted. There are two ways to optimize those mesh regions:

   - run several times the adaptation in order to cancel the non-adaptation of the interfaces. To this end, the following call is used:

     ```
     shrimp -O 5 -in name -out final -mem 512 -bucket 512 -ncut 8
     -nproc 8 -sref -nloop 2
     ```

     `-nloop` states the number of optimization step for the whole adaptive algorithm. After each adaptation the mesh is gathered and then re-split for a new adaptation.

   - One may separate the point insertion phase from the mesh optimization phase in the adaptive process. In order to separate these phases, the option `-soptim` is used. The full command line becomes:

     ```
     shrimp -O 5 -in name -out final -mem 512 -bucket 512 -ncut 8
     -nproc 8 -sref -nloop 2 -soptim
     ```

4 Modules and options overviews

To execute `Shrimp` the following syntax is used:

`shrimp`

With no parameter, `Shrimp` prints the standard command line help message:

```
usage: shrimp -O [n] [-v[n]] [-h] [opts..] -in filein[.mesh] [-sol filesol[.sol]] [-out fileout[.sol]]

** Module options :
  -O 1  : Reorder mesh  : Cache misses reduction, optimized indirect addressing
  -O 2  : Split mesh    : for parallel runs
  -O 3  : Gather meshes : collect meshes for serial runs
  -O 4  : Parallel mesh adaptation
  -O 5  : Perform modules 2 --> 4 --> 3 nloop times

** Inline options :
  -ncut        : number of components for the splitting
  -nordm       : no random for the splitting
  -nocor       : no correction to ensure connex partition
  -nloop       : number of iteration for module 5

** Options passed to mesh generator :
  -mem     ival    : memory
  -bucket  ival    : bucket
  -soptim          : Split  mesh generation/optimization
                     Use module 1 before optimization step

** Options for parallel runs
  -dir   sval  : remote directory (default is current directory)
  -host        : read file host.dat and use hosts to run in parallel

** Generic options :
  -sref     : Re-code references on short int
  -tref     : Use existing tetrahedra reference for partition
  -f        : Save fileout in ascii
  -f32/-f64 : Force 32 or 64 bits real numbers
  -h        : Print this message
  -bsc      : output mesh in bsc format
  -w        : Write host.dat file
  -v   ival : Tune level of verbosity
  -o   ival : Tune level of outputs

  -in  string : in file name
  -out string : out file name
  -sol string : initial solution file name
```

The standard use of the different modules has been explained in the section Howto. We give now the list of generic options related to the I/O that have not been described:

`-f`	Save output files in ascii. By default, `Shrimp` write the output files in binary format.
`-f32`	Save output files in 32-bits real number. By default, `Shrimp` write in 64-bits (double precision).
`-f64`	Force `Shrimp` to save output files in 64-bits real number.
`-h`	Print the help in the terminal.
`-o [int]`	Tune the files output level.
`-v [int]`	Tune the level of verbosity
`-w`	Write the `host.dat` file.

The input files names and the output files are given with the following option:

`-in [char]`	Specify the input file names for the mesh and the solution.
`-sol [char]`	Specify the solution input file name, if it is different from the mesh file name.
`-out [char]`	Specify the output file name for the mesh and the solution. If not specified, the output file name is the input file name concatenated with "`.o`".

Let us give some examples:

```
shrimp -O 1 -in name
```

It reads `name.mesh[b]` and `name.sol[b]`. It writes `name.o.mesh[b]` and `name.o.sol[b]`.

```
shrimp -O 1 -in name -sol solname
```

It reads `name.mesh[b]` and `solname.sol[b]`. It writes `name.o.mesh[b]` and `name.o.sol[b]`.

```
shrimp -O 1 -in name -sol solname -out outnam
```

It reads `name.mesh[b]` and `solname.sol[b]`. It writes `outname.mesh[b]` and `outname.sol[b]`.

5 Some application examples

5.1 A few words about Hilbert curves

Hilbert Curves is an example of space filling curves introduced in the last century to study cardinality properties of real spaces. More precisely, they were used to prove that the cardinality of a square is equal to the cardinality of one of its sides. Examples of curves approximating the continuous Hilbert curve are depicted by Figure 1. In numerical applications, a 3D computational domain need to be map onto the memory of a computer which is one-dimensional. Hilbert curves are one method to achieve this mapping. Hilbert curves are used in `shrimp` for two different tasks that aims at achieving high performance computing:

- reduce cache misses and increase speed of serial code,
- perform adaptive parallel mesh adaptation by providing a simple, fast and robust mesh partitioning strategy for anisotropic meshes.

Cache misses are due to indirect addressing, see Figure 2 (left). They occur when data are required for a computation and those data are not available in the current cache line. It is worth mentioning that the cost of a cache miss is far more important that classical operations used in numerical applications: mutilply, divide, multadd, ... Figure 2 (right) depicts the CPU cycles of a cache miss compared to the cost of classical operations. As regards cache misses reduction, we use some compactness properties of Hilbert curves. It mainly ensures that points closed in the space are also closed on the curve, and consequently, closed in the memory after reordering, cf. Figure 1.

As a mapping exists from a 3D domain onto this curve, splitting the 1D domain is equivalent to split the 3D domain. This simple idea is at the basis of the mesh partitioning strategy derived in `Shrimp`. On simple geometries that are uniformly meshed, this strategy leads to very natural partitions, see Figure 3. This is no more the case when dealing with highly anisotropic meshes, see Figure 4.

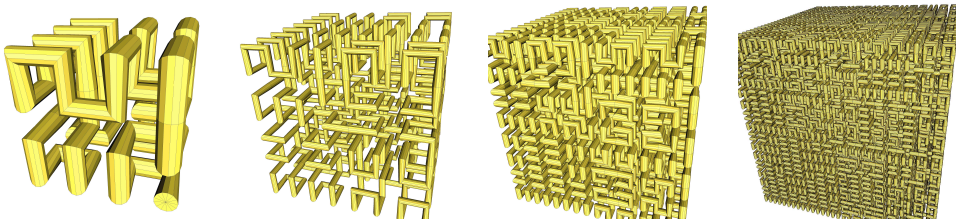


Figure 1: Several discrete Hilbert curves of a cube. The yellow line represents the path of the Hilbert curve. The limit to these curves is the continuous Hilbert curve that fills the cube.

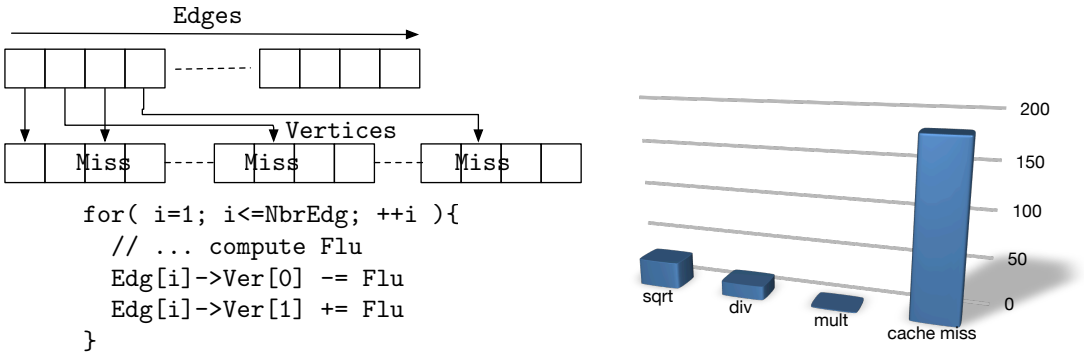


Figure 2: Left, example of a typical indirect addressing loop. The edge loop requires vertices information to pursue the computation. When the required vertices is not directly available, a cache miss occurs. Right, comparison of required CPU cycles between classical numerical operations and a cache miss on a Mac Intel 64bits architecture.

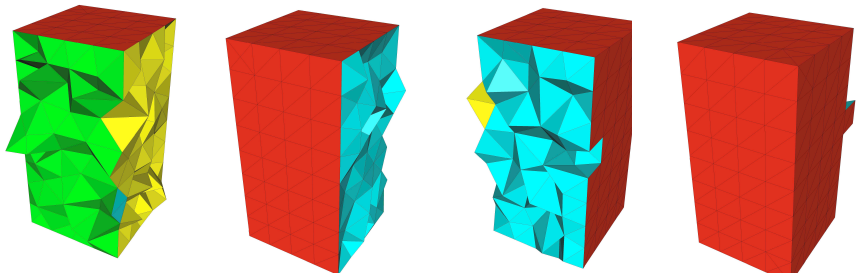


Figure 3: Example of Hilbert-based partition of a cube into 4 parts.

All the presented examples below are in the context of mesh adaptation. Therefore, a small theoretical background on error estimates, unit mesh, and metric tensor, may be needed for comprehensiveness. Details about mesh adaptation can be found in the following references [1, 5]. The examples described hereafter come from our local benchmark data base. Some of the test cases are depicted in Figure 5. The data base is composed of uniform, adapted isotropic and adapted anisotropic meshes. The range of the number of vertices varies from 10 000 to 10 000 000.

5.2 Speeding up serial codes

We consider some examples of numerical simulations. Speed-ups are given for the flow solver `Wolf` and for the local adaptive mesh generator `Mmg3d`. All the timings include of course the

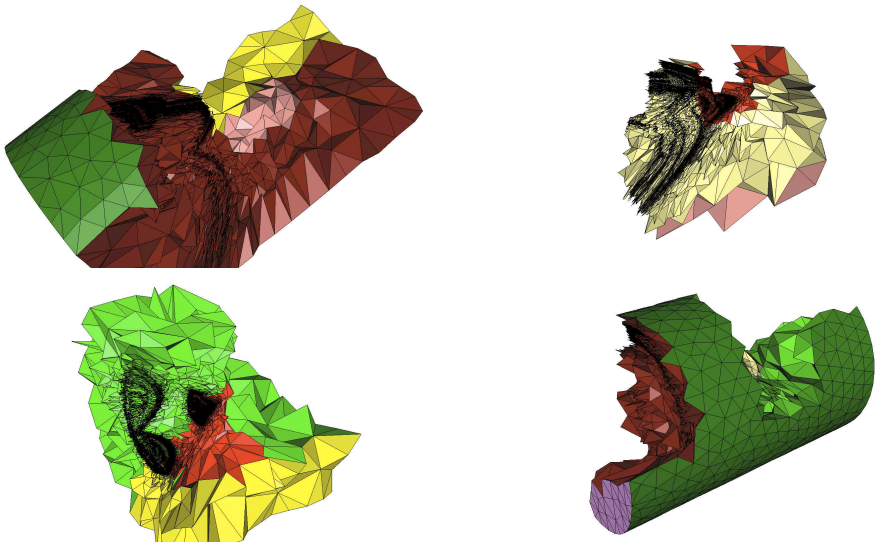


Figure 4: Several Hilbert-based partitions arising from an initial highly anisotropic mesh.

Entities sort	1.3	to	1.5
+ Hilbert sort	2.5	to	3

Table 1: Range of speed-ups obtained for `Wolf` with only the entities sorted and with the entities sorted coupled with the Hilbert renumbering. These speed-ups include reordering time.

time for reordering. All runs are done in serial. This section only involves the module 1 of `Shrimp`.

In most of the test cases, the serial codes are at least twice faster when the Hilbert renumbering strategies is used, see Figures 6 and 7. The impact is even stronger on the mesh generator. If we compare the number of vertices inserted by second, one may see the benefit of the Hilbert reordering, see Figure 8. Indeed, when the mesh generator inserts new points, the proximity in space and in memory can vary, especially if the new points are stored at the end of the vertices array.

5.3 Parallel mesh adaptation

`Shrimp` is used conjointly with `Mmg3d` on a daily basis to perform parallel adaptive mesh generation. A bunch of examples can be found in [5]. These examples involve the module 5 that corresponds to a sequential call of modules $2-4-3$ n-times, where n is set with the option `-nloop`. For the input, we assume that we have a mesh supplied with a metric field

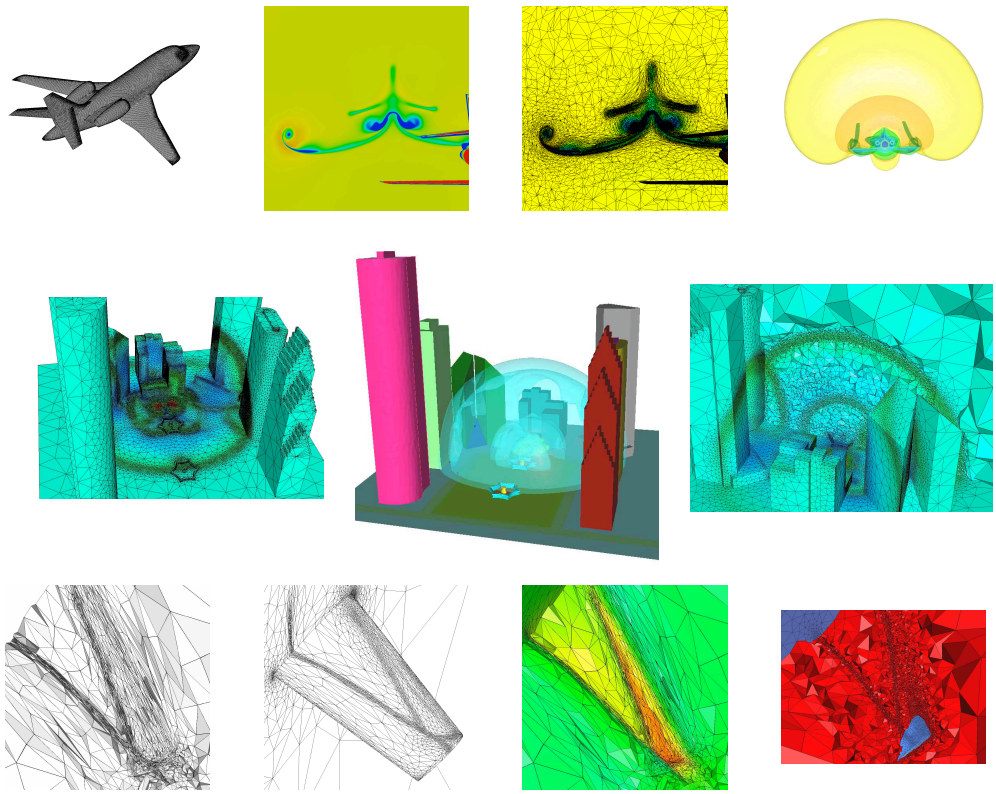


Figure 5: Some examples of meshes used to provide quantitative information on the performance of `Shrimp`. Examples are quoted from [1, 5].

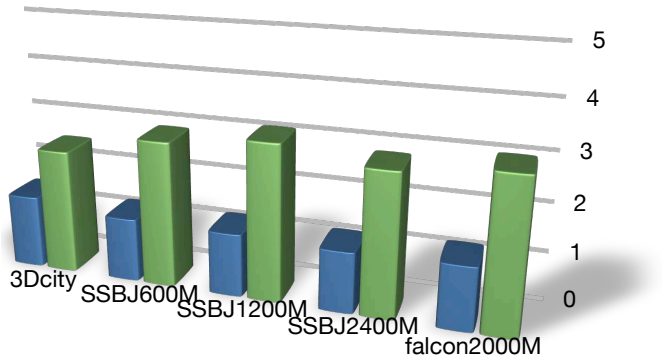


Figure 6: Speed-ups obtained for the flow solver `Wolf` with the Hilbert renumbering for each test case of the benchmark data base. In each case, the renumbered version of the serial code is at least twice faster that the original serial code. These speed-ups include the reordering time.

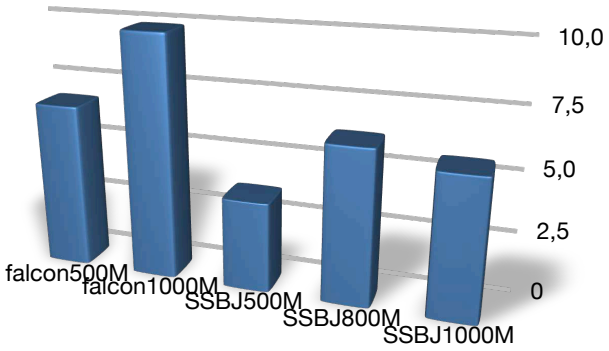


Figure 7: Speed-ups obtained for adaptive mesh generator `Mmg3d` with the Hilbert renumbering for each test case of the benchmark data base. In each case, the renumbered version of the serial code is at least twice faster that the original serial code. These speed-ups include the reordering time.

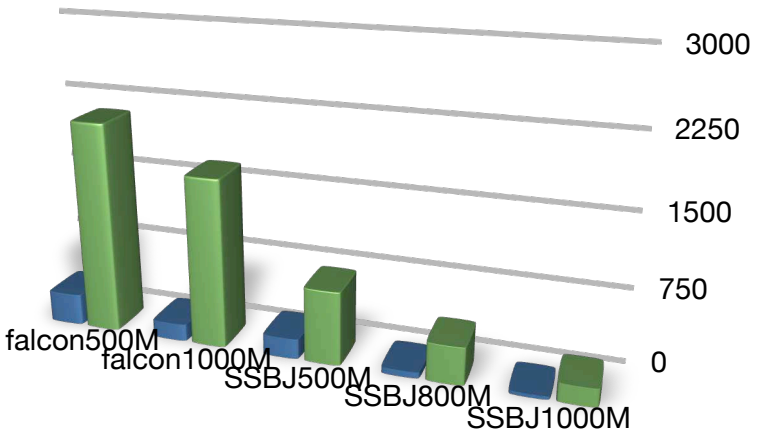


Figure 8: Number of vertices inserted per second for adaptive mesh generator `Mmg3d` without (blue) and with (green) the Hilbert renumbering for each test case of the benchmark data base. The benefit of the Hilbert reordering is clearly illustrated.

defined at its vertices. We aim at generating a unit mesh with respect to the input metric tensors field. All the examples are run on a Mac personal computer equipped with a Intel Core 2 at 2.8 GHz and 15Gb of memory.

First example. We consider a supersonic flow around a spike. The case comes from an experimental simulation [4] carried out at NASA Langley. The partitioning algorithm is timed on the final anisotropic adapted mesh composed of 8 069 621 vertices, 182 286 boundary triangles and 48 045 800 tetrahedra, see Figure 9. The timings of each step of the domain decomposition algorithm are:

- Reading input data: 61.136s
- Create an initial Hilbert partition: 107.617s
- Create neighboring structure: 33.329s
- Hashing boundary faces: 0.009s

The outputted partitions are well weighted if we look at the number of tetrahedra:

Partition	1	6 005 546
Partition	2	6 005 699
Partition	3	6 006 002
Partition	4	6 005 590
Partition	5	6 005 721
Partition	6	6 005 802
Partition	7	6 005 681
Partition	8	6 005 759

The time to create the 8 connex partitions and to write the corresponding meshes is about 116s. The maximal memory allocated in this case is about 5.6Gb. The complete step is done in 280s. The mesh partition size variation is less than 0.008%.

As regards the partitions gathering, the algorithm is very low memory consuming as only the interfaces of the meshes are stored. For this example, the complete gathering step requires 40s and 65Mb of memory.

Second example. A supersonic flow around a complex aircraft is considered. The geometry of the aircraft is depicted in Figure 10. The mesh is composed of 9 083 531 vertices, 555 650 boundary triangles and 53 884 863 tetrahedra. Note that this example has a large number of facets contrary to the previous spike geometry. The timings of each step of the domain decomposition algorithm are:

- Reading input data: 58.922s
- Create an initial Hilbert partition: 99.777s
- Create neighboring structure: 37.311s
- Hashing boundary faces: 0.029s

The outputted partitions are well weighted if we look at the number of tetrahedra:

Partition	1	6 734 843
Partition	2	6 736 148
Partition	3	6 735 775
Partition	4	6 735 544
Partition	5	6 736 143
Partition	6	6 725 021
Partition	7	6 745 562
Partition	8	6 735 827

The time to create the 8 connex partitions and to write the corresponding meshes is about 120.488s. The maximal memory allocated in the case is about 6.3Gb. The complete step is done in 280s. The mesh partition size variation is less than 0.15%. It is higher than the previous example. In fact, the good balancing of partitions depends on the complexity of the geometry.

The complete gathering step requires 30s and 80Mb of memory.

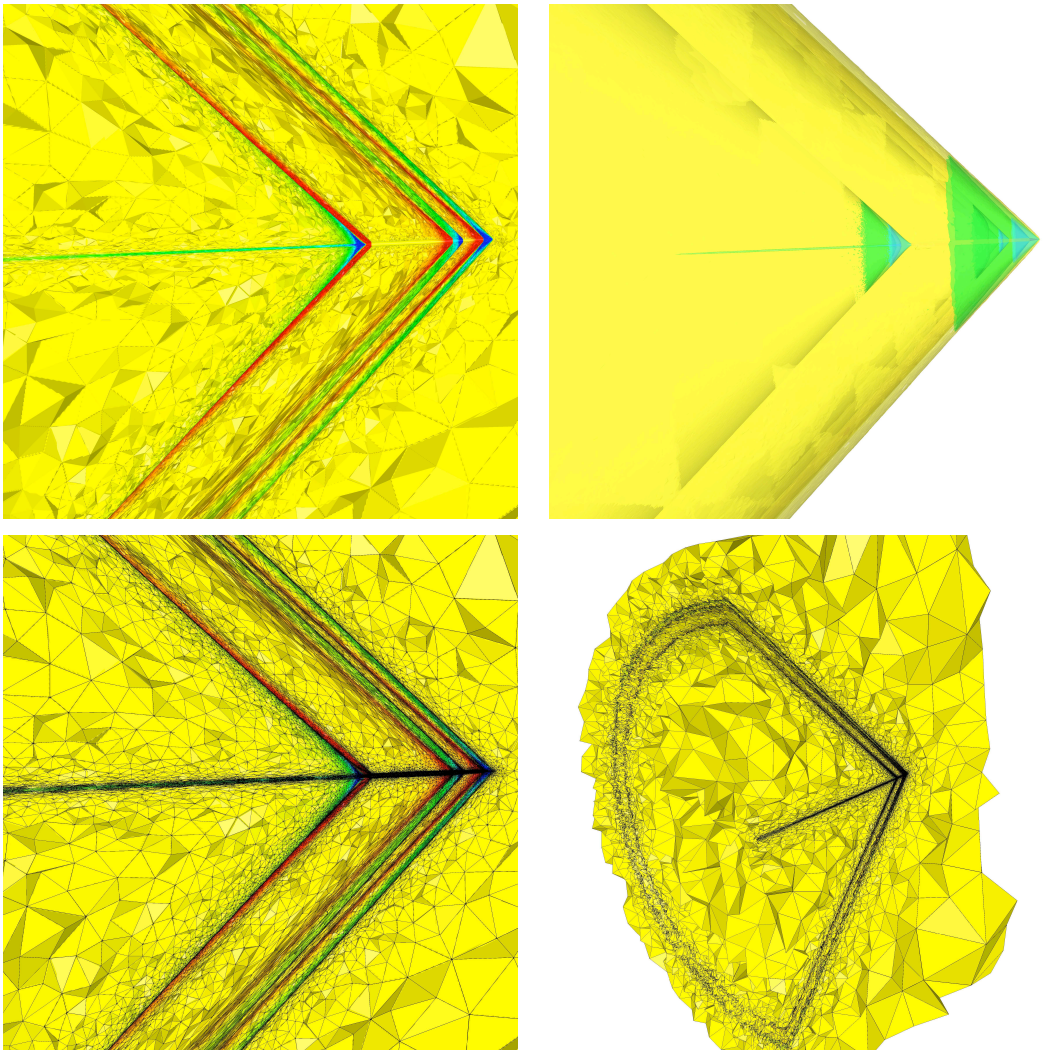


Figure 9: Anisotropic mesh and final solution for the spike test case. Top right, iso-values of the Mach number.

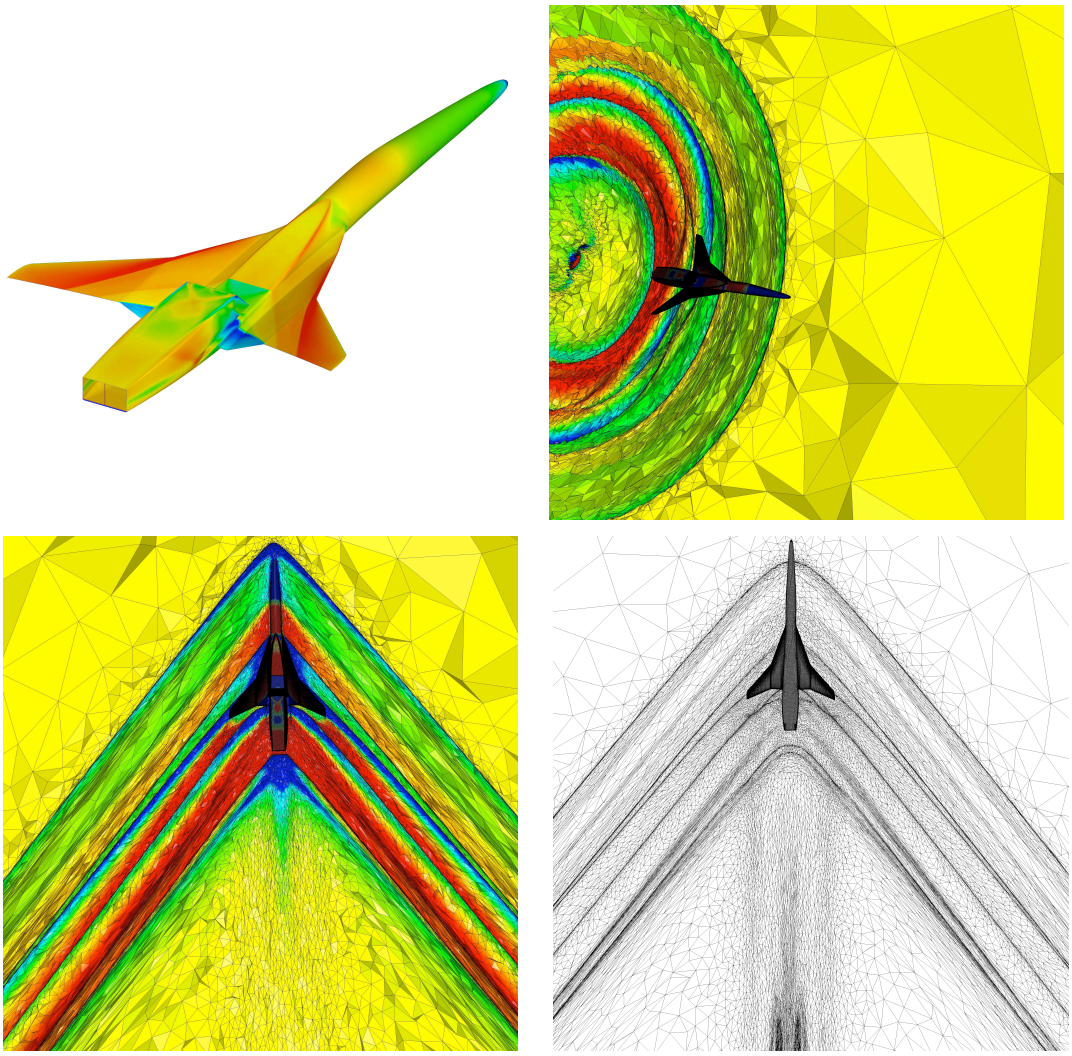


Figure 10: Final anisotropic adapted mesh and solution for the supersonic aircraft.

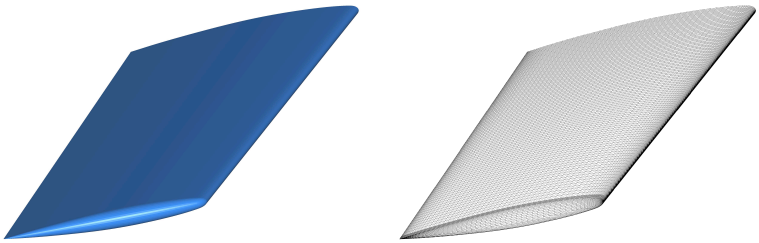


Figure 11: ONERA M6 wing geometry (left) and initial surface mesh (right).

5.4 User defined mesh partitioning

Mesh partitioning can also be monitored by the user. `Shrimp` handles references at tetrahedra to create and to correct (if needed or requested) the partitions. It corresponds to the module 2 where the option *-tref* is activated through the command line.

As an example, we consider a transonic viscous flow around an ONERA M6 wing, see Figure 11. In this kind of application, the flow features impose to have two distinct parts in the mesh: the boundary layer and the inviscid mesh. Generally, metric-based mesh adaptation for the inviscid part is perfomed in order to capture accurately shocks. However, keeping a semi-structured boundary layer mesh is necessary to capture the viscous effect of the flow in the boundary layer. In that case, `Shrimp` can be used to separate the viscous mesh from the inviscid one. The Euler mesh can be then adapted separately. This domain decomposition based on tetrahedron references (tags) is exemplified Figure 12.

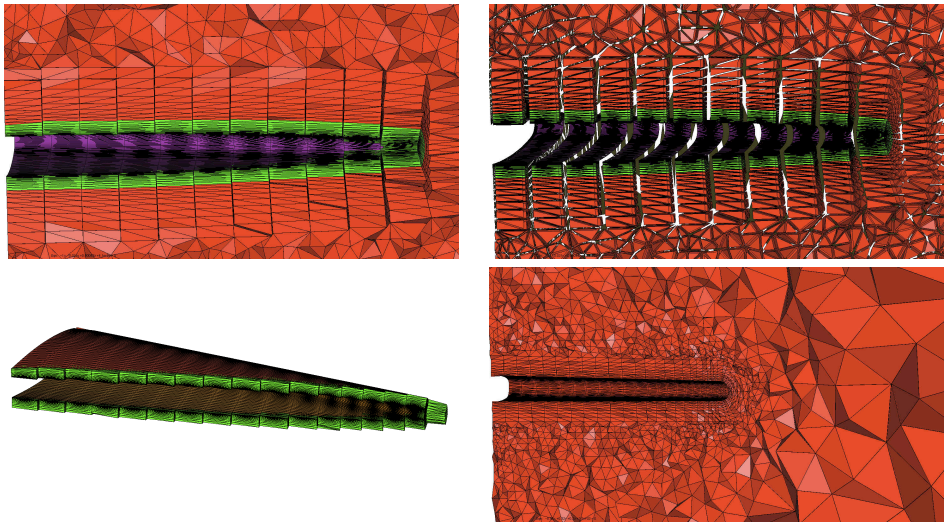


Figure 12: Top, initial mesh where the lower part of the boundary layer mesh is tagged differently. Bottom, the two final partitions: left, the wing surface and the corresponding layers of semi-structured mesh, right, the inviscid mesh.

References

[1] F. Alauzet. *Adaptation de maillage anisotrope en trois dimensions. Application aux simulations instationnaires en Mécanique des Fluides*. PhD thesis, Université Montpellier II, Montpellier, France, 2003.

[2] F. Alauzet and A. Loseille. High order sonic boom modeling by adaptive methods. RR-6845, INRIA, February 2009.

[3] C. Dobrzynski and P. J. Frey. Anisotropic delaunay mesh adaptation for unsteady simulations. In *Proc. of 17th Int. Meshing Rountable*, pages 177–194. Springer, 2008.

[4] H. W. Carlson and R. J. Mack and O. A. Morris. A wind-tunnel investigation of the effect of body-shape on sonic-boom pressure distributions. TN. D-3106, Nasa, 1965.

[5] A. Loseille. *Adaptation de maillage 3D anisotrope multi-échelles et ciblé à une fonctionnelle. Application à la prédiction haute-fidélité du bang sonique.* PhD thesis, Université Pierre et Marie Curie, Paris VI, Paris, France, 2008.

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Futurs : Parc Club Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex (France)
Unité de recherche INRIA Lorraine : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)
Unité de recherche INRIA Rennes : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier (France)
Unité de recherche INRIA Sophia Antipolis : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-0803